Iekkë: A SAT-Based Bounded-Round Verifier for Multi-Threaded Programs (Competition Contribution)

Paolo Di Biase[1,5] * **, Bernd Fischer[2], Salvatore La Torre[3], Peter Schrammel[4,6], and Gennaro Parlato[5]

[1] Gran Sasso Science Institute (GSSI), L'Aquila, Italy, paolo.dibiase@gssi.it
[2] Stellenbosch University, Stellenbosch, South Africa, bfischer@cs.sun.ac.za
[3] University of Salerno, Fisciano (SA), Italy, slatorre@unisa.it
[4] University of Sussex, Brighton, UK, p.schrammel@sussex.ac.uk
[5] University of Molise, Pesche, Italy, gennaro.parlato@unimol.it
[6] Diffblue Ltd, UK

Abstract. Iekkë is a sequentialization-inspired bounded model checking (BMC) tool for verifying multi-threaded C programs. It is derived from Deagle and reuses the CBMC front-end for parsing, goto-program construction, loop unwinding, and SSA-based formula generation. Iekkë differs from both CBMC and Deagle in its treatment of shared-memory interactions: it replaces the classic partial-order memory constraints with a *bounded-round* encoding inspired by lazy sequentialization. The resulting shared-memory constraints are *linear* in the number of memory events of the bounded program and linear in the round bound k, avoiding the quadratic/cubic growth typical of partial-order encodings. Because the concurrency constraints are compact and purely propositional, Iekkë relies on an off-the-shelf SAT solver (e.g., MiniSAT) rather than a dedicated ordering-theory solver. Iekkë supports reachability, assertion and data-race checking, and produces SV-COMP witnesses for violations.

Keywords: Program verification · Bounded model checking · Concurrency · SAT.

1 Verification Approach

Background. For shared-memory concurrency, BMC tools must connect reads and writes across threads in a way that corresponds to a consistent execution. Classical encodings, including the partial-order approach used in CBMC, introduce constraints over event orders (e.g., happens-before, coherence) and enforce consistency by global properties such as acyclicity [1,12]. On memory-intensive instances, these constraints can dominate the formula size.

* Jury member/Corresponding author.
** Work performed while at the University of Molise.

S. Junges and G. Katz (Eds.): TACAS 2026, LNCS 16506, pp. 536–542, 2026.
https://doi.org/10.1007/978-3-032-22749-2_29

Bounded-round semantics. Iekkë (pronounced *ˈjɛkkə*) under-approximates the interleaving space by restricting the analysis to executions that can be partitioned into at most k *rounds*. Intuitively, in each round threads execute in a fixed global order; each thread executes a (possibly empty) contiguous segment of its SSA steps and then yields to the next thread. This scheduling discipline is inspired by lazy sequentialization (e.g., Lazy-CSeq) [9,11] and is effective for bug finding because many concurrency bugs manifest within a small number of context-switches [13]. Lazy-CSeq's sequentialization is also used in [8,10,5,14,15].

Bounded-round memory constraints. For a fixed k, Iekkë introduces variables that represent (i) the per-thread *context-switch points* delimiting the segments executed by the thread in each round, and (ii) the *round assignment* of memory events. Shared-memory writes update the per-round snapshots of the shared variables; shared-memory reads select the value of the latest visible write according to the round and thread order. Crucially, the number of additional constraints is *linear* in (a) the number of shared-memory events in the bounded program and (b) the round bound k.

Relation to Lazy-CSeq and CBMC/Deagle. Iekkë is designed to inherit the strengths of both Lazy-CSeq and CBMC while addressing key weaknesses of each. From Lazy-CSeq, Iekkë inherits the idea of bounding concurrency by a small number of rounds and the resulting compact treatment of shared-memory interactions. However, unlike source-level lazy sequentialization, Iekkë does *not* replicate each thread's code for every round. Instead, it builds on CBMC's SSA-based front-end and introduces round/context-switch-point variables directly at the formula level, keeping a *single* symbolic copy of the program while still restricting schedules to at most k rounds. From CBMC, Iekkë inherits the mature parsing, unwinding, and SSA generation pipeline, but it replaces CBMC's resp. Deagle's partial-order-style shared-memory constraints—which can grow quadratically or worse in the number of events—with its own bounded-round memory constraints that are linear in the number of shared-memory events and in the round bound k. CBMC and Deagle both support weak memory models while Iekkë currently supports only sequential consistency.

Why SAT suffices. Deagle tries to address the scalability issues of large partial-order encodings by integrating a specialized solver component to reason about ordering consistency [6,7]. In contrast, Iekkë reduces the blow-up at the source: bounded-round constraints make shared-memory consistency largely explicit in the schedule (i.e., round and thread order), yielding a compact propositional encoding that can be handled effectively by standard CDCL SAT solvers such as MiniSAT [4]. Moreover, the round bound k provides a smooth trade-off between compactness and coverage. In the extreme, choosing k large enough (e.g., $k \geq |E|$, where E is the set of shared-memory events in the bounded program) allows Iekkë to simulate executions with as many context-switch opportunities as there are memory events, and thus to capture the same read/write interaction patterns as the existing CBMC/Deagle encodings on the bounded program. In this worst

case, the size of the shared-memory constraints becomes $O(|E| \cdot k) = O(|E|^2)$, still avoiding the cubic growth that can arise in certain partial-order formulations.

Supported properties. Iekkë supports all reachability checks supported by CBMC (e.g., assertion violations, `unreach-call`) and concurrency-specific checks. For data races, Iekkë searches for executions witnessing conflicting unsynchronized accesses and reports them as violations.

2 Software Architecture

Implementation lineage. Iekkë inherits from but is not built directly on top of CBMC. Instead, it is derived from Deagle, which already incorporates several concurrency-oriented optimizations in the encoding of thread functions (partly inherited from Yogar-CBMC). Iekkë reuses this optimized CBMC/Deagle infrastructure for parsing, unwinding, and SSA generation, but replaces Deagle's partial-order/ordering-theory layer with our bounded-round shared-memory constraints. Iekkë's pipeline thus comprises the following steps:

- **Front-end.** This step includes preprocessing and parsing of C; construction of a goto-program; modeling of common concurrency primitives (e.g., Pthreads); loop unwinding up to the user-provided bound; and the SSA generation for thread-local control/data flow [12].
- **Concurrency encoding.** This step replaces Deagle's partial-order layer with new bounded-round shared-memory constraints. The encoding is generated natively at the formula level (rather than via source-to-source sequentialization), which enables a tight integration with the existing SSA pipeline.
- **Solving.** The resulting propositional formula is discharged by an off-the-shelf SAT solver; we use MiniSAT as default solver.
- **Counterexamples and witnesses.** Iekkë includes in the constructed SSA form the mapping between the SSA variables and the original program variables, following the style of CBMC/Deagle. Thus, for violated properties, Iekkë can reuse the counterexample generation and trace reporting infrastructure of the CBMC/Deagle tool chain.

Iekkë is implemented in `C++`, given its CBMC-derived architecture and codebase.

3 Strengths and Weaknesses

Strengths. On unsafe benchmarks where violations occur within a few rounds, Iekkë is competitive while often using substantially less time and memory than partial-order approaches such as Deagle and CBMC. On benchmarks with many memory events, the linear growth in events and rounds avoids the cubic blow-up typical of classic partial-order encodings, improving practical scalability.

Weaknesses. Iekkë uses a BMC approach; in addition to the bounds on loop unwinding and recursion depth, it introduces a bound on the number of rounds. In contrast to the former two, Iekkë currently does not automatically increase the round bound, which must be specified by the user; if a bug requires more rounds for discovery than specified, the tool will not report it. The approach is thus an under-approximation and is more specialized to find bugs.

Results. Iekkë scored 3428 points, earning the bronze medal in the *Concurrency* category [2]. It identified 308 bugs with confirmed witnesses and 216 without. Iekkë gave incorrect answers for 44 safe benchmarks due to minor implementation bugs and misclassified 22 unsafe benchmarks as safe: 3 of these were caused by Deagle issues, while the rest were due to considering only 3 rounds. Deagle correctly solved 40 of these 66 benchmarks. Iekkë performed well overall, except in the `no-data-race` sub-category, solving only 646 out of 1030 benchmarks.

4 Tool Setup and Configuration

Installation. Iekkë installation is identical to that of Deagle; detailed instructions are provided in the repository `README`.

Parameters. Iekkë reuses CBMC/Deagle's standard parameters (i.e., architecture bit-width, unwinding, property selection). The maximum number of computation rounds is set via `--lazy-c-seq-rounds`; in the competition, we simply use a fixed value $k = 3$, since real bugs often manifest within 3 rounds; however, we can improve this by an iterative deepening scheme. Property selection is via dedicated parameters. For example, the `no-overflow` property requires the additional parameters `--no-assertions`, `--signed-overflow-check`, and `--unsigned-overflow-check`. We refer to the Zenodo archive for further details.

Participation. Iekkë participates only in the *Concurrency* category. We support the following properties: `no-data-race`, `no-overflow`, `unreach-call`, and `valid-memsafety`.

Usage. A typical invocation is:

```
deagle_exe file.c --lazy-c-seq-rounds 3 --unwind 9 <other opts>
```

5 Software Project

Iekkë is developed by the authors based on Deagle 4.1.0 and its corresponding CProver/CBMC infrastructure, using MiniSAT as default SAT backend. Iekkë is licensed under GPLv3; see the repository for the exact competition package, third-party licenses, and build instructions.

Acknowledgements

We thank the developers and maintainers of CBMC, MiniSAT, and Deagle for making their tools available and for their contributions to the verification community. This work was supported by the Erasmus+ International Credit Mobility Project 2024-1-IT02-KA171-HED-000224869, by INDAM-GNCS 2024-2025, FARB 2022–25 grants of Università degli Studi di Salerno, and by AWS Amazon Research Awards. Paolo Di Biase was supported by a Research Scholarship (Borsa di Ricerca) from the University of Molise.

Data Availability Statement

The tool is open-source and developed in the Lazy-PO repository:

`https://github.com/paolo-di-biase/Lazy-PO`

The SV-COMP submission is based on commit 50ddac2. The competition snapshot is also archived on Zenodo [3].

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 141–157. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
3. Di Biase, P., Fischer, B., La Torre, S., Schrammel, P., Parlato, G.: Iekkë artifact. Zenodo (2025). https://doi.org/10.5281/zenodo.17708300
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
5. Fischer, B., Garbi, G., La Torre, S., Parlato, G., Schrammel, P.: Static data race detection via lazy sequentialization. In: Castañeda, A., Enea, C., Gupta, N. (eds.) Networked Systems - 12th International Conference, NETYS 2024, Rabat, Morocco, May 29-31, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14783, pp. 124–141. Springer (2024). https://doi.org/10.1007/978-3-031-67321-4_8
6. He, F., Sun, Z., Fan, H.: Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 1264–1279. ACM (2021). https://doi.org/10.1145/3453483.3454108

7. He, F., Sun, Z., Fan, H.: Deagle: An smt-based verifier for multi-threaded programs (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 424–428. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
8. Inverso, O., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. pp. 807–812. IEEE Computer Society (2015). https://doi.org/10.1109/ASE.2015.108
9. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 585–602. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_39
10. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 398–401. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
11. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. ACM Trans. Program. Lang. Syst. **44**(1), 1:1–1:50 (2022). https://doi.org/10.1145/3478536
12. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. CoRR **abs/2302.02384** (2023). https://doi.org/10.48550/ARXIV.2302.02384
13. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785
14. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 753–764. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115686
15. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 193–200. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886679